ECE459 Lab 2

Jackson Guo
j84guo@uwaterloo.ca

1 Common Logic

The message passing and shared memory solutions share common logic. Both solutions' high-level control flow is listed below.

Table 1. Common control flow

Step	Description
1	The main thread parses command line arguments to determine the JWT token's message and signature.
2	A pool of worker threads is started. The main thread generates all possible secrets and uses some communication mechanism to assign these candidate secrets to the workers. In the message passing solution, the communication mechanism is a bounded Crossbeam channel. In the shared memory solution, it is a bounded queue protected by two semaphores and a mutex.
3	Workers check each of the candidate secrets provided to them by the main thread. If the answer is found, the answer is communicated to the main thread. In the message passing solution, the answer is communicated using a bounded Crossbeam channel of size one. In the shared memory solution, an atomic boolean (an answer has been found) and a mutex-protected `Option<String>` (what the answer is) are used.
4	The main thread stops producing new candidate secrets and indicates to the workers that they should finish. In both solutions, the master does this by sending a special value through the same mechanism that was used to communicate candidate secrets. Workers which receive this special value know they should return.
5	The main thread joins all worker threads and then prints the answer which was found, or an error message if none was found.

1.1 Candidate secret generation

The provided sample code performed generated candidate secrets using a recursive depth-first search, where a candidate secret is the direct parent of any others which extend it by one character, e.g. “a” is the parent of “aa”, “ab”, “ac” and so on.

Both final solutions choose to do this iteratively instead - an iterative depth-first search avoids the stack frame creation and teardown which is necessary for each recursive call, noticeably speeding up candidate secret generation.

```
fn generate_secrets(alphabet: &[u8],
                    max_len: usize,
                    /* Channel or shared queue */) {
    /* Candidate secrets sent in batches to reduce communication overhead */
    const BATCH_SIZE: usize = 1 << 5;
    let mut batch = Vec::<Vec<u8>>::new();

    /* Depth-first search stack */
    let mut stack = vec![Vec::<u8>::new()];
    while stack.len() > 0 {
        if !res_recv_end.is_empty(/* Answer found by any worker */) {
            return;
        }

        /* Current “node” */
        let secret = stack.pop().unwrap();
        batch.push(secret.clone());
        if batch.len() == BATCH_SIZE {
            /* Send batch into channel or shared queue, then reset batch */
            batch = Vec::<Vec<u8>>::new();
        }

        /* Push “children” onto stack if we haven't reached the max length */
        if secret.len() < max_len {
            for c in alphabet {
                let mut next_sec = secret.clone();
                next_sec.push(*c);
                stack.push(next_sec);
            }
        }
    }

    if batch.len() > 0 {
        /* Send batch into channel or shared queue */
    }
}
```

Note that candidate secrets are sent in batches of size 32 into the channel or shared queue - this reduces communication overhead while still allowing work to be distributed relatively evenly between workers.

Furthermore, this function returns as soon as any worker signals that it found an answer.

1.2 Starting worker threads and joining them

After parsing command line arguments, the main thread creates one worker thread for each available virtual CPU - having more isn't expected to be helpful since this task is CPU-intensive.

```
let num_workers = num_cpus::get();
```

The worker threads are then started and their join handles collected.

```
let workers = start_consumers(num_workers,
                              &message,
                              &signature,
                              /* Channel or shared state used by workers */);
```

Secret generation is initiated - as per step 4, this returns if a worker signals that an answer has been found. Regardless of whether an answer was found, the main thread sends a special value to each worker telling it to return - all worker threads are joined.

Note that in the case where an answer is found, the main thread does not need to wait for the entire solution space to be searched - rather, it just needs to wait until the threads deque and process whatever candidate secrets are currently in the bounded channel/queue.

```
/* Either way, tell all workers to stop */
for _ in 0..num_workers {
   /* Send special values indicating workers should return */
}

/* Join workers */
for w in workers {
   w.join().unwrap();
}
```

Finally, the main thread prints the answer (or not, if one wasn't found).

```
if /* Check for answer */ {
   println!("No answer found");
} else {
   let answer = /* Obtain answer as Vec<u8> */
   println!("{}", std::str::from_utf8(&answer).unwrap());
}
```

2 Message Passing

In the message passing solution, one channel is created to send candidate secrets to the workers and another is used by a worker to signal that it found an answer.

```
/* Let buffer capacity be a multiple of number of workers */
let buffer_capacity = num_workers * 4;

/* Channel for sending secrets from the main thread to workers */
let (sec_send_end, sec_recv_end) =
bounded::<Option<Vec<Vec<u8>>>>(buffer_capacity);

/* Channel for a worker to signal that it has found the answer */
let (res_send_end, res_recv_end) = bounded::<Vec<u8>>(1);
```

The main thread sends candidate secrets into the secret channel.

```
/* Called from function generate_secrets) */
sec_send_end.send(Some(batch)).unwrap();
```

Worker threads repeatedly receive from the secret channel until an answer is found, or the main thread indicates workers should return.

```
/* Start a worker */
thread::spawn(move || {
    loop {
        let batch = match sec_recv_end.recv().unwrap() {
            Some(batch) => batch,
            /* Main thread wants workers to return */
            None => return
        };
        for secret in batch {
            if is_secret_valid(&message, &signature, &secret) {
                /* Signal that the answer has been found */
                res_send_end.try_send(secret).unwrap();
                return;
            }
        }
    }
})
```

Once secret generation returns, the main thread sends `None` values to tell worker threads they should return, and then joins them (join loop omitted since it was shown in section 1.2).

```
for _ in 0..num_workers {
   sec_send_end.send(None).unwrap();
}
```

Finally, the main thread checks the channel of size 1 to see if an answer was found.

```
/* Check for answer */
if res_recv_end.is_empty() {
   println!("No answer found");
} else {
   let answer = res_recv_end.recv().unwrap();
   println!("{}", std::str::from_utf8(&answer).unwrap());
}
```

3 Shared Memory

In the shared memory solution, a bounded shared queue is used instead of a channel to communicate secrets.

```
#[derive(Clone)]
struct SharedBuffer {
   buffer: Arc<Mutex<VecDeque<Option<Vec<Vec<u8>>>>>>,
   items: Arc<Semaphore>,
   spaces: Arc<Semaphore>
}
```

The shared queue is protected by two semaphores and a mutex as shown in lecture 11 - one semaphore indicates the number of items, another indicates the number of spaces, and the mutex serializes concurrent queue push/pop operations.

```
fn push(&self, x: Option<Vec<Vec<u8>>>) {
   let permit = block_on(self.spaces.acquire());
   {
       let mut buffer = self.buffer.lock().unwrap();
       buffer.push_back(x);
   }
   self.items.add_permits(1);
   permit.forget();
}

fn pop(&self) -> Option<Vec<Vec<u8>>> {
   let permit = block_on(self.items.acquire());
   let x = {
       let mut buffer = self.buffer.lock().unwrap();
       buffer.pop_front().unwrap()
   };
   self.spaces.add_permits(1);
   permit.forget();
   return x;
}
```

In addition to a shared buffer for candidate secrets, an atomic bool is used by a worker to tell the main thread that an answer has been found. The answer itself is communicated through a mutex-protected `Option<String>`.

```
let buffer_capacity = num_workers * 4;
let shared_buffer = SharedBuffer::new(buffer_capacity);
let is_answer_found = Arc::new(AtomicBool::new(false));
let answer = Arc::new(Mutex::<Option<String>>::new(None));
```

Worker threads repeated pop from the shared queue until an answer is found, or the main thread indicates they should return.

```
thread::spawn(move || {
    loop {
        let batch = match shared_buffer.pop() {
            Some(batch) => batch,
            None => return
        };
        for secret in batch {
            if is_secret_valid(&message, &signature, &secret) {
                let s = std::str::from_utf8(&secret).unwrap().to_string();
                {
                    *answer.lock().unwrap() = Some(s);
                }
                is_answer_found.store(true, Ordering::SeqCst);
                return;
            }
        }
    }
})
```

The main thread stops generating secrets as soon as the atomic bool is enabled (join loop omitted). It then tells the workers to return by pushing `None` values.

```
for _ in 0..num_workers {
    shared_buffer.push(None);
}
```

Finally, the main thread checks for an answer stored in the mutex-protected `Option<String>`.

```
{
    match answer.lock().unwrap().deref() {
        Some(answer) => println!("{}", answer),
        None => println!("No answer found")
    };
}
```

4 Output Correctness

The following test cases were run. Both solutions found the correct answer for each one, provided the maximum length was long enough - an error was reported if the maximum length was less than the secret's length.

Test Case	JWT Token	Secret
1	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJPbmxpbmUgSldUIEJ1aWxkZXIiLCJpYXQiOjE1OTQxNjE5NjIsImV4cCI6MTYyNTY5Nzk2MiwiYXVkIjoid3d3LmV4YW1wbGUuY29tIiwic3ViIjoianJvY2tldEBleGFtcGxlLmNvbSIsIkdpdmVuTmFtZSI6IkpvaG5ueSIsIlN1cm5hbWUiOiJSb2NrZXQiLCJFbWFpbCI6Impyb2NrZXRAZXhhbXBsZS5jb20iLCJSb2xlIjpbIk1hbmFnZXIiLCJQcm9qZWN0IEFkbWluaXN0cmF0b3IiXX0.dOqyg4rFU_QoSdKqOsmOXIVrTgygsT2AzDHL8gMZ17E	a
2	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJPbmxpbmUgSldUIEJ1aWxkZXIiLCJpYXQiOjE1OTQxNjE5NjIsImV4cCI6MTYyNTY5Nzk2MiwiYXVkIjoid3d3LmV4YW1wbGUuY29tIiwic3ViIjoianJvY2tldEBleGFtcGxlLmNvbSIsIkdpdmVuTmFtZSI6IkpvaG5ueSIsIlN1cm5hbWUiOiJSb2NrZXQiLCJFbWFpbCI6Impyb2NrZXRAZXhhbXBsZS5jb20iLCJSb2xlIjpbIk1hbmFnZXIiLCJQcm9qZWN0IEFkbWluaXN0cmF0b3IiXX0.KpPBh8EWP0D965dNUscAnc2Q0Vb1-WLWegk_acd6rZI	aaa
3	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJPbmxpbmUgSldUIEJ1aWxkZXIiLCJpYXQiOjE1OTQxNjE5NjIsImV4cCI6MTYyNTY5Nzk2MiwiYXVkIjoid3d3LmV4YW1wbGUuY29tIiwic3ViIjoianJvY2tldEBleGFtcGxlLmNvbSIsIkdpdmVuTmFtZSI6IkpvaG5ueSIsIlN1cm5hbWUiOiJSb2NrZXQiLCJFbWFpbCI6Impyb2NrZXRAZXhhbXBsZS5jb20iLCJSb2xlIjpbIk1hbmFnZXIiLCJQcm9qZWN0IEFkbWluaXN0cmF0b3IiXX0.PcrNM34a2vyKzJDnpi0h792zNuPchSKf4M577nhHnWE	05re
4	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.shNrMqeoWA3La5bOmJ9rzGtX8rh4M9fR93HVbE3JQTA	0123
5	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9	01234

	lIiwiaWF0IjoxNTE2MjM5MDIyfQ.2fQveXQ-Af9-aI4lwINSON5WN1EyLDqFoHCxd3hnGjA	

5 Memory Leaks and Data Races

No memory leaks (i.e. not freed and pointer lost) are reported by Valgrind for any solution and test case.

Helgrind reported possible data races for both solutions related to code in the sha2 crate used for secret checking. Specifically, when atomics are used.

```
==26158== Possible data race during read of size 1 at 0x3CF010 by thread #3
==26158== Locks held: none
==26158==    at 0x15FDB6: core::sync::atomic::atomic_load (atomic.rs:2355)
==26158==    by 0x15FEDF: core::sync::atomic::AtomicU8::load (atomic.rs:1486)
==26158==    by 0x15ECA6: cpuid_bool::LazyBool::unsync_init (lib.rs:42)
==26158==    by 0x15ADB1: sha2::sha256::x86::compress (x86.rs:101)
==26158==    by 0x155040: sha2::sha256::compress256 (sha256.rs:168)
==26158==    by 0x154C5C: sha2::sha256::Engine256::update::{{closure}} (sha256.rs:33)
==26158==    by 0x15E7FF: block_buffer::BlockBuffer<BlockSize>::input_blocks (lib.rs:81)
==26158==    by 0x154C0D: sha2::sha256::Engine256::update (sha256.rs:33)
==26158==    by 0x11FC91: <sha2::sha256::Sha256 as digest::Update>::update (sha256.rs:70)
==26158==    by 0x12176D: <hmac::Hmac<D> as crypto_mac::NewMac>::new_varkey (lib.rs:157)
==26158==    by 0x125EC0: shared_mem::is_secret_valid (shared-mem.rs:25)
==26158==    by 0x133B46: shared_mem::start_consumers::{{closure}} (shared-mem.rs:118)
==26158==
==26158== This conflicts with a previous write of size 1 by thread #5
==26158== Locks held: none
==26158==    at 0x15FE71: core::sync::atomic::atomic_store (atomic.rs:2341)
==26158==    by 0x15FF2E: core::sync::atomic::AtomicU8::store (atomic.rs:1514)
==26158==    by 0x15ECF0: cpuid_bool::LazyBool::unsync_init (lib.rs:45)
==26158==    by 0x15ADB1: sha2::sha256::x86::compress (x86.rs:101)
==26158==    by 0x155040: sha2::sha256::compress256 (sha256.rs:168)
==26158==    by 0x154C5C: sha2::sha256::Engine256::update::{{closure}} (sha256.rs:33)
==26158==    by 0x15E7FF: block_buffer::BlockBuffer<BlockSize>::input_blocks (lib.rs:81)
==26158==    by 0x154C0D: sha2::sha256::Engine256::update (sha256.rs:33)
==26158==  Address 0x3cf010 is 0 bytes inside data symbol
"_ZN4sha26sha2563x868compress10CPUID_BOOL17h13e8f71950edaca4E"
```

It also complained about tokio's block_on function which I use to acquire semaphores. Again, atomics seem to be the issue.

```
==26158== Possible data race during write of size 1 at 0x5E16AB8 by thread #5
==26158== Locks held: none
==26158==    at 0x15FE64: core::sync::atomic::atomic_store (atomic.rs:2340)
```

```
==26158==    by 0x12F454: core::sync::atomic::AtomicBool::store (atomic.rs:414)
==26158==    by 0x12C9EE: futures_executor::local_pool::run_executor::{{closure}} (local_pool.rs:100)
==26158==    by 0x1169C3: std::thread::local::LocalKey<T>::try_with (local.rs:272)
==26158==    by 0x1168C2: std::thread::local::LocalKey<T>::with (local.rs:248)
==26158==    by 0x12C835: futures_executor::local_pool::run_executor (local_pool.rs:83)
==26158==    by 0x12CA70: futures_executor::local_pool::block_on (local_pool.rs:317)
==26158==    by 0x12631A: shared_mem::SharedBuffer::pop (shared-mem.rs:57)
==26158==    by 0x13388B: shared_mem::start_consumers::{{closure}} (shared-mem.rs:113)
==26158==    by 0x11EE31: std::sys_common::backtrace::__rust_begin_short_backtrace
(backtrace.rs:125)
==26158==    by 0x12F04F: std::thread::Builder::spawn_unchecked::{{closure}}::{{closure}} (mod.rs:474)
==26158==    by 0x115A6F: <std::panic::AssertUnwindSafe<F> as
core::ops::function::FnOnce<()>>::call_once (panic.rs:322)
==26158==
==26158== This conflicts with a previous read of size 1 by thread #1
==26158== Locks held: none
==26158==    at 0x16863A: core::sync::atomic::atomic_swap (atomic.rs:2372)
==26158==    by 0x1684C4: core::sync::atomic::AtomicBool::swap (atomic.rs:443)
==26158==    by 0x1688DA: <futures_executor::local_pool::ThreadNotify as
futures_task::arc_wake::ArcWake>::wake_by_ref (local_pool.rs:63)
==26158==    by 0x168863: futures_task::arc_wake::ArcWake::wake (arc_wake.rs:32)
==26158==    by 0x16AAF1: futures_task::waker::wake_arc_raw (waker.rs:49)
==26158==    by 0x14A8FA: core::task::wake::Waker::wake (wake.rs:218)
==26158==    by 0x14AF0F: core::ops::function::FnMut::call_mut (function.rs:150)
==26158==    by 0x147C8C: core::iter::traits::iterator::Iterator::for_each::call::{{closure}} (iterator.rs:675)
==26158==  Address 0x5e16ab8 is 24 bytes inside a block of size 32 alloc'd
==26158==    at 0x4C30F2F: malloc (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==26158==    by 0x16933B: alloc::alloc::alloc (alloc.rs:84)
==26158==    by 0x1693F9: alloc::alloc::Global::alloc_impl (alloc.rs:164)
==26158==    by 0x169679: <alloc::alloc::Global as core::alloc::AllocRef>::alloc (alloc.rs:224)
==26158==    by 0x16929C: alloc::alloc::exchange_malloc (alloc.rs:314)
==26158==    by 0x168A97: alloc::sync::Arc<T>::new (sync.rs:330)
==26158==    by 0x168968: futures_executor::local_pool::CURRENT_THREAD_NOTIFY::__init
(local_pool.rs:54)
==26158==    by 0x168358: core::ops::function::FnOnce::call_once (function.rs:227)
==26158==    by 0x167AA8: std::thread::local::lazy::LazyKeyInner<T>::initialize (local.rs:304)
==26158==    by 0x169A15: std::thread::local::fast::Key<T>::try_initialize (local.rs:473)
==26158==    by 0x169CBA: std::thread::local::fast::Key<T>::get (local.rs:456)
==26158==    by 0x1689BC: futures_executor::local_pool::CURRENT_THREAD_NOTIFY::__getit
(local.rs:183)
```

Upon further research, it appears that Helgrind only understands POSIX synchronization primitives. From the docs:

Make sure your application, and all the libraries it uses, use the POSIX threading primitives ...

Do not roll your own threading primitives (mutexes, etc) from combinations of the Linux futex syscall, atomic counters, etc. These throw Helgrind's internal what's-going-on models way off course and will give bogus results.

As it turns out, Tokio's semaphores and Crossbeam's channels do not use POSIX primitives. They're implemented in terms of atomic booleans, futexes on Linux, i.e. things which make Helgrind complain. Therefore, I think it is highly likely my errors from Helgrind are actually false positives.

6 Benchmarking

Benchmarking was done with hyperfine. The shared memory and message passing programs had similar average times across all test cases. For test case 1, they were slightly faster than the sequential lab2 program.

For test case 2, the multi-threaded programs were both considerably slower than the sequential solution. The reason for this is that the secret "aaa" is encountered early (relative to the size of the solution space) when generating candidate secrets starting from "" as described in section 1.1. While the sequential program is able to exit immediately once it finds the answer, the multi-threaded solutions must finish processing any candidate secrets currently enqueued in the channel or shared buffer before the main thread can exit - this extra tear down time is what causes the multi-threaded solutions to be slower for this test case.

For test cases 3-5 the multi-threaded programs were nearly 7x-8x faster than the sequential program. In these test cases, the answer was deep enough in the solution space that the tear down time is small compared to the time for a worker to find the answer. The improvement of 7x-8x reflects the 8 virtual CPUs available to the multi-threaded programs - the sequential program was only running on 1. The improvement is not quite 8x due to some of the program being sequential in nature (i.e. command line parsing, shared buffer initialization) as well as startup, synchronization, and teardown overhead in the parallel portion of the program (i.e. secret generation and validation).

The table below shows the average times taken by each program for the test cases.

Table 2. Average time by each program for each test case

Test case	shared-mem average time (ms)	message-passing average time (ms)	lab2 average time (ms)
1	1.1	2	2.4
2	53.2	58.6	2.6
3	532.5	612.4	3.891

4	327.1	306.1	2.461
5	10.96	11.223	85.717

The bar graph below shows the table data on a logarithmic scale.

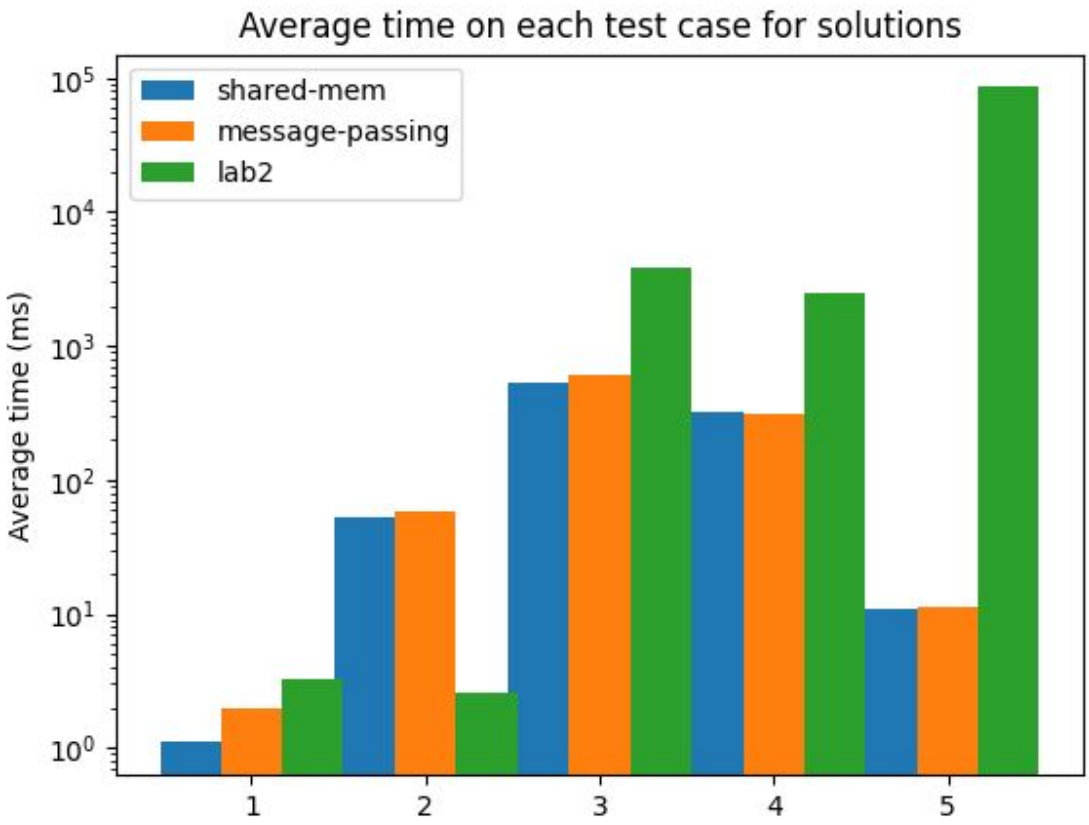

Figure 1. Average time by each program for each test case (logarithmic scale)